PRÁCTICA 1 - VARIABLES COMPARTIDAS

Ejercicio 1

Para el siguiente programa concurrente suponga que todas las variables están inicializadas en 0 antes de empezar. Indique cual/es de las siguientes opciones son verdaderas:

a. En algún caso el valor de x al terminar el programa es 56.

b. En algún caso el valor de x al terminar el programa es 22.

c. En algún caso el valor de x al terminar el programa es 23.

P1::

```
if (x = 0) then
   y:= 4*2;
   x:= y + 2;
```

P2::

```
if (x > 0) then
 x:= x + 1;
```

P3::

```
x:= (x*3) + (x*2) + 1;
```

Respuestas

a. Verdadero

b. Verdadero

c. Verdadero

Ejercicio 2

Realice una solución concurrente de grano grueso (utilizando <> y/o <await B; S>) para el siguiente problema. Dado un número N verifique cuántas veces aparece ese número en un arreglo de longitud M. Escriba las pre-condiciones que considere necesarias.

Respuesta

```
int cantidad = 0; int buscado = N; int array[M];

Process Buscador[id: 0 ... M-1] {
    if (array[id] == buscado) {
        <cantidad ++;>
    }
}
```

Ejercicio 3

Dada la siguiente solución de grano grueso:

a. Indicar si el siguiente código funciona para resolver el problema de Productor/Consumidor con un buffer de tamaño N. En caso de no funcionar, debe hacer las modificaciones necesarias.

```
int cant = 0; int pri_ocupada = 0; int pri_vacia = 0; int buffer[N];

Process Productor:: {
    while (true) {
        //produce elemento
        <await (cant < N); cant++>
        buffer[pri_vacia] = elemento;
        pri_vacia = (pri_vacia + 1) mod N;
    }
}

Process Consumidor:: {
    while (true) {
        <await (cant > 0); cant-- >
        elemento = buffer[pri_ocupada];
        pri_ocupada = (pri_ocupada + 1) mod N;
        //consume elemento
 }
}
```

b. Modificar el código para que funcione para C consumidores y P productores.

Respuestas

a. Puede ocurrir que el consumidor consuma un elemento que no existe. Esto ocurre porque solo se ejecuta de forma atómica el aumento de la cantidad cuanto el productor produce un elemento, si luego de hacer eso pierde

el uso del procesador y lo toma el consumidor, este puede consumir un elemento que no existe ya que el productor no lo ha guardado en el buffer.

```
int cant = 0; int pri_ocupada = 0; int pri_vacia = 0; int buffer[N];

Process Productor:: {
    while (true) {
        //produce elemento
        <await (cant < N); cant++;
        buffer[pri_vacia] = elemento;>
        pri_vacia = (pri_vacia + 1) mod N;
    }
}

Process Consumidor:: {
    while (true) {
        <await (cant > 0); cant--;
        elemento = buffer[pri_ocupada];>
        pri_ocupada = (pri_ocupada + 1) mod N;
        //consume elemento
 }
}
```

b.

```
int cant = 0; int pri_ocupada = 0; int pri_vacia = 0; int buffer[N];

Process Productor[id: 0 ... P-1] {
    while (true) {
        //produce elemento
        <await (cant < N); cant++;
        buffer[pri_vacia] = elemento;
        pri_vacia = (pri_vacia + 1) mod N;>
    }
}

Process Consumidor[id: 0 ... C-1] {
    while (true) {
        <await (cant > 0); cant--;
        elemento = buffer[pri_ocupada];
        pri_ocupada = (pri_ocupada + 1) mod N;>
        //consume elemento
 }
}
```

Ejercicio 4

Resolver con SENTENCIAS AWAIT (<> y <await B; S>). Unsistema operativo mantiene 5 instancias de un recursoalmacenadas en una cola, cuando un proceso necesita usaruna instancia del recurso la saca de la cola, la usa ycuando termina de usarla la vuelve a depositar.

Respuesta

```
ColaRecursos cola;
int  cantRecursosDisponibles = 5;

Process Proceso[id: 0 ... N-1] {
    Recurso recurso;
    <await (cantRecursosDisponibles > 0);
    recurso = cola.pop();
    cantRecursosDisponibles--;>
    // uso del recurso
    <cola.push(recurso);
    cantRecursosDisponibles++;>
}
```

Ejercicio 5

En cada ítem debe realizar una solución concurrente degrano grueso (utilizando <> y/o <await B; S>) para el siguiente problema, teniendo en cuenta las condiciones indicadas en el item. Existen N personas que deben imprimir un trabajo cada una.

a. Implemente una solución suponiendo que existe una única impresora compartida por todas las personas, y las mismas la deben usar de a una persona a la vez, sin importar el orden. Existe una función Imprimir(documento) llamada por la persona que simula el uso de la impresora. Sólo se deben usar los procesos que representan a las Personas.

b. Modifique la solución de (a) para el caso en que se deba respetar el orden de llegada.

c. Modifique la solución de (a) para el caso en que se deba respetar el orden dado por el identificador del proceso (cuando está libre la impresora, de los procesos que han solicitado su uso la debe usar el que tenga menor identificador).

d. Modifique la solución de (b) para el caso en que además hay un proceso Coordinador que le indica a cada persona que es su turno de usar la impresora.

Respuestas

a.

```
Process Persona[id: 0 ... N-1] {
    Documento documento;
    <Imprimir(documento);>
}
```

b.

```
ColaPersonas cola;
int siguiente = -1;

Process Persona[id: 0 ... N-1] {
    Documento documento;
    <if (siguiente == -1) siguiente = id;
    else cola.push(id);>
    <await (siguiente == id);>
    Imprimir(documento);
    <if (cola.isEmpty()) siguiente = -1;
    else siguiente = cola.pop();>
}
```

c. Asumo la existencia de la función **pushOrdenado(id)** que se encarga de insertar el id en la cola de manera ordenada.

```
ColaPersonas cola;
int siguiente = -1;

Process Persona[id: 0 ... N-1] {
    Documento documento;
    <if (siguiente == -1) siguiente = id;
    else cola.pushOrdenado(id);>
    <await (siguiente == id);>
    Imprimir(documento);
    <if (cola.isEmpty()) siguiente = -1;
    else siguiente = cola.pop();>
}
```

d.

```
ColaPersonas cola;
int siguiente = -1;
boolean impresoraOcupada = false;

Process Persona[id: 0 ... N-1] {
    Documento documento;
    <cola.push(id);>
    <await (siguiente == id);>
    Imprimir(documento);
    impresoraOcupada = false;
}

Process Coordinador {
    while true {
        <await (!impresoraOcupada && !cola.isEmpty());>
        siguiente = cola.pop();
        impresoraOcupada = true;
    }
}
```

Ejercicio 6

Dada la siguiente solución para el Problema de la Sección Crítica entre dos procesos (suponiendo que tanto SC como SNC son segmentos de código finitos, es decir que terminan en algún momento), indicar si cumple con las 4 condiciones requeridas:

```
int turno = 1;

Process SC1:: {
    while (true) {
        while (turno == 2) skip;
        SC;
        turno = 2;
        SNC;
    }
}

Process SC2:: {
    while (true) {
        while (turno == 1) skip;
        SC;
        turno = 1;
        SNC;
    }
}
```

Respuesta

Yo creo que se cumplen todas las propiedades menos la de "Ausencia de Demora Innecesaria" que se cumple parcialmente ya que, si el proceso SC2 siempre le gana al SC1 en el uso del Procesador y termina siendo el único proceso ejecutándose, nunca va a poder entrar a la sección crítica porque necesita que primero entre SC1 para que cambie el turno.

Ejercicio 7

Desarrolle una solución de grano fino usando sólo variables compartidas (no se puede usar las sentencias await ni funciones especiales como TS o FA). En base a lo visto en la clase 3 de teoría, resuelva el problema de acceso a sección crítica usando un proceso coordinador. En este caso, cuando un proceso SC[i] quiere entrar a su sección crítica le avisa al coordinador, y espera a que éste le dé permiso. Al terminar de ejecutar su sección crítica, el proceso SC[i] le avisa al coordinador. Nota: puede basarse en la solución para implementar barreras con “Flags y coordinador” vista en la teoría 3.

Respuesta

```
int N = ...; int actual = -1;
bool peticiones[N];

Process Worker[i = 1 to N] {
    while (true) {
        peticiones[i] = true;
        while (actual != i) skip;
        // Sección Crítica
        actual = -1;
    }
}

Process Coordinador {
    while (true) {
        for [i = 1 to N] {
            if (peticiones[i]) {
                peticiones[i] = false;
                actual = i;
                while (actual != -1) skip;
            }
        }
    }
}
```